Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1940 – 1949

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Debugging Scientific Applications With Statistical Assertions

Minh Ngoc Dinh[a,*], David Abramson[a], Jin Chao[a], Donny Kurniawan[a]

Andrew Gontarek[b], Bob Moench[b], Luiz DeRose[b]

[a]*Monash University, Clayton, 3800, Victoria, Australia*
[b]*Cray Inc, 380 Jackson St # 210, St Paul, MN, 55101-2987, United States*

Abstract

Traditional debuggers are of limited value for modern scientific codes that manipulate large complex data structures. Current parallel machines make this even more complicated, because the data may be distributed across multiple processors, making it difficult to view, interpret and validate the contents of a distributed structure. As a result, many applications' developers resort to placing validation and display code directly in the source program itself. This paper discusses a novel debug-time assertion, called a "Statistical Assertion", that allows a user to reason about large data structures. We present the design and implementation of statistical assertions, and illustrate the debugging technique with a molecular dynamics simulation. We evaluate the performance of the system on a 12,000 processor Cray XE6, and show that it is useful for real time debugging.

Keywords: Debugging; Assertion; Statistic; Parallel Architecture

1. Introduction

Many scientific codes manipulate enormous multi-dimensional data structures, often distributed across parallel processes, and it is impractical for a user to trace problems by focussing on individual data elements. Our earlier work has demonstrated that ad hoc debug-time assertions can assist in this task because it is not necessary to examine every value in a large data structure [1]. The study showed that in a number of cases, a parallel computer can be used to execute the assertion logic, making it efficient when used on large data structures and machines.

This paper introduces a new type of ad hoc debug-time assertion called a *Statistical Assertion*. Such assertions allow users to reason about derived metrics rather than the raw data. The essence of this approach is to (1) diminish the substantial amount of raw data to manageable "blocks"; (2) alleviate the complexity in managing data decomposition across a large number of processors; and (3) leverage parallelism to make the system fast enough for real time debugging. Statistical assertions can test the 'statistical' state of a large distributed array, and can be refined iteratively by the user in order to locate the source of an error. Because evaluation is likely to be expensive, we propose a scheme that executes the assertion in parallel, making assertions over large data structures feasible.

* Corresponding author. Tel.: +61 3 9903 4145.
E-mail address: nmdin1@student.monash.edu.

doi:10.1016/j.procs.2012.04.212

We also discuss how partial statistical results are aggregated, and compared. These ideas are implemented in an existing parallel debugger, Guard [1, 2]. The rest of the paper demonstrates sample use of the proposed techniques in debugging a molecular dynamics application, and presents performance results from a 12,000 core Cray XE6.

2. Motivation and Statistical Assertions

As more data is produced and gathered, *statistic* and *data patterning* (or *data mining*) [3] is becoming an increasingly important concept for transforming data into *information*. For example, data mining is popular in a wide range of profiling practices, such as marketing, finance, climate modelling, and earth systems [3]. In addition, most highly performance software, not only generates raw data, but also produces patterning information in forms of histograms, probability distributions, or data models (i.e mathematical functions). These statistics not only give the users great insights of the observed phenomena, but also sometimes display unusual details of the computation.

Recently, we recognized the importance of extracting statistical information for debugging purposes whilst chasing an error in one of our software tools (the debugger, in fact). Specifically, we generated a performance model based on a set of simulations, and produced a plot that summarized the model behavior with a two-dimensional graph. This simple display highlighted an error, and we subsequently found a coding bug. Importantly, the error became obvious not through the *detailed examination of process state*, as supported by almost all debugging tools, but through a simple proxy – a graph showing one derived variable against another. The location of the bug could be deduced quite accurately without viewing the source code, because the graph contained sufficient information about the type of error. This example highlights the potential of using statistic instead of raw data to locate coding defects.

A *statistical assertion* is defined as a predicate consisting of two *data models* in the form of either statistical primitives (e.g. mean or standard deviation values) or data models (e.g. histograms or density functions). Statistical assertions allow the user to compare data pattern information between two data structures, instead of comparing the exact values like the assertions used in previous work [1, 2]. For example, it is possible to assert that the *mean* value of a large dataset is between certain bounds during the life of a function call, or the number of elements in an array needs to be in a specific range. More advanced statistical assertions allow the user to state that the *histogram* formed by all elements in a specific dataset must be equivalent to a *histogram* formed by another dataset, or assert that all elements in dataset should be normally distributed.

Statistical assertions are useful in debugging large scale scientific problems because (1) they allow users to focus on the scientific *meaning* of the computations instead of the exact data values produced by them, and (2) they reduce the complexity in debugging *stochastic processes,* for example as found in Monte Carlo methods. Statistics can be used to reflect scientific knowledge behind a computation, thus by using statistical assertions; a user can integrate such knowledge into the debugging process and transform it into runtime invariants that ensure the correct execution of the code at runtime. Failure to comply with such expectations will lead the users to the incorrect computation. Furthermore, stochastic processes pose a nontrivial difficulty in testing and debugging, because the program state is often nondeterministic. However, if we disregard the exact data values, the data patterns extracted from those datasets are often deterministic. Statistical assertions allow the users to capture such determinism, and make debugging stochastic simulations more practical, whilst reducing the complexity of processing raw data.

3. Design of Statistical Assertions

The support for statistical assertions requires a framework that addresses two issues. First, it needs to support a wide range of useful statistics, and it is desirable to compute these in parallel in order to provide real time debugging of large datasets. Second, the framework should allow users to create arbitrary user-defined data models. The following texts focus on these issues respectively. Details regarding the implementation are discussed in section 4.

3.1. Split-phase Statistical Operation

The parallel computation of basic statistics such as average, max, min is relatively straight forward; however, more complex statistics require special handling. For example, given a dataset X, the typical standard *two-pass algorithm* for computing standard deviation [3] is not efficient in parallel, because it requires all elements in X to be examined twice. Even though there are one-pass algorithms that compute the standard deviation value, some of them

are numerically unstable. However, the *pair-wise algorithm* [3] provides much better accuracy. Given X=A∪B, one can compute the variance in one pass by computing sample mean value μ and the *sum of squares of differences* from μ, denoted as $M_{2,i}$, for A and B independently. These values can later be combined to calculate the overall standard deviation value using the following formulas

$$M_{2,X} = M_{2,A} + M_{2,B} + \delta^2 \frac{n_A n_B}{n_X} \quad (1)$$

$$stdev = \sqrt{\frac{M_{2,X}}{n_X}} \quad (2)$$

This algorithm can be executed in two phases, and the first phase can be parallelised (as depicted in Figure 1). Therefore, we design the statistical reduction process as a *split-phase* operation, as below. As it turns out, this design actually maps well onto the client/server architecture of our host debugger, as discussed in the section 4 [1].

Parallel: calculate a set of primary statistics from the input dataset. Examples are sample size, minimum, maximum, mean etc. This phase is embarrassingly parallel as it does not involving any inter-process communication; and

Aggregation: assemble a collection of primary statistics to form a full statistical model. This contains both primary and derived statistics, for instance variance or standard deviation values.

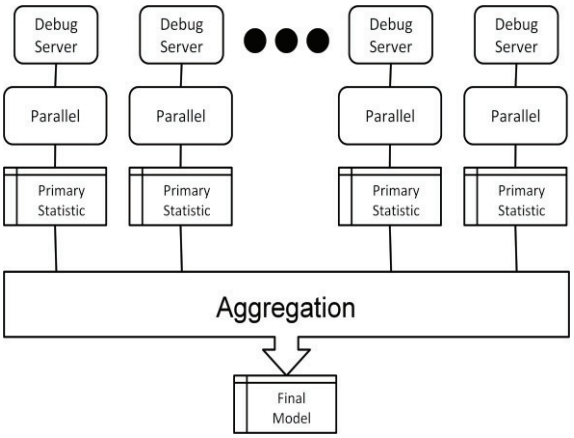

Figure 1 - Split-phase Statistical Operation

3.2. User-defined Abstract Data Models

Some statistical assertions require users to create arbitrary data models. For example, to assert that elements in a dataset follow a Gaussian distribution, the histogram constructed using the target dataset must be compared against a histogram built with random numbers, generated from the Gaussian distribution. In order to create accurate distributions, we need to perform quite a few computations, and it is desirable to parallelise this operation as well. The split-phase scheme presented above can be used to create distributions in parallel.

4. Implementation

To test the ideas proposed in section 3, we have implemented statistical debug-time assertions in Guard, an existing parallel command line debugger [1]. In this section, we present the implementation details.

4.1. User-defined Statistic Function

There are a few options for allowing users to create statistical reduction functions. First, a debugger command language can be introduced. For example, TotalView defines a special scripting language called *TVScript* [6]. Another approach is to integrate an existing scripting language (and associated runtime), such as Python [7], into the debugger. Finally, it is possible to leverage a conventional programming language, such as C, and its compiler and runtime, by compiling arbitrary modules and linking these into the debugger binary. This approach has the disadvantage that erroneous library code can crash the debugger itself; however, it significantly simplifies the implementation of a prototype. Accordingly, we have used this technique here. To make it easier for a user to write statistical functions, we have defined an API that standardises the interface. Further, we have implemented a set of pre-defined statistics using the same technique, and these act as templates for users wishing to develop their own functions. Thus, Guard comes with functions such as min/max, element counting, standard deviation, variance, and so forth, implemented as externally defined and compiled library, and these can be extended as required.

Function Template

To enforce the split-phase statistic framework discussed in section 3.1, a general template is provided in which a few compulsory functions are expected. Consider the pseudo code below:

```
func my_func (data)
    define my_func_server (data) = server_result;
    define my_func_client (collection) = client_result;
end
```

Here, *my_func* is defined as a parallel reduction function, and the user must define two sub-functions: *my_func_server* and *my_func_client*. *my_func_server* specifies the computation that can be performed on sub-structures retrieved from the back-end debug servers. This function represents the first phase, the parallel phase, of the split-phase mechanism discussed in section 3.1. *my_func_client* specifies how the results of the server function are collected from debug servers, merged and transformed (e.g *client_result*) during assertion evaluation.

Compilation and Deployment

After coding, users are not required to compile the code. A debugger command called *register* compiles the source and links it executable against the provided API to create a shared library object. The function name (e.g *my_func*) is stored in the debugger function table, and the share library object is shipped to a location which is accessible to both remote debug servers and the front-end client. More details about how debug server and debug client can identify which user-defined function to invoke during runtime are discussed later in section 4.3.1.

Example

Here, we consider an example where the standard deviation operation can be specified in Guard. Note that the pair-wise algorithm is just illustrative of what can be achieved using user-defined functions, and how such functions can be parallelised. We chose this for the initial implementation because it is efficient. Alternatively, the traditional two-pass algorithm for calculating standard deviation value could be implemented with the existing architecture, with the extra overhead.

```
func stdev_server(a[], n) {
    for i from 1 to n {   sum += a[i];  }     result.mean = sum / n;      result.size = n;
    for i from 1 to n {   result.sum2 += sqr(a[i] - result.mean);  }     return result;      }
func stdev_client(servers[], num_procs) {
    cur_mean=servers[1].mean;  cur_size=servers[1].size;  cur_sum2=servers[1].sum2;
    for i from 2 to num_procs    {
        sigma=servers[i].mean−cur_mean;
        cur_mean=cur_mean+sigma*servers[i].size/(cur_size+servers[i].size);
        cur_sum2=cur_sum2+servers[i].sum2+sigma*sigma*cur_size*servers[i].size/(cur_size+ servers[i].size);
        cur_size=servers[i].size + cur_size;  }  result = sqrt (cur_sum2/size);   return result        }
```

The pseudo code above defines two functions: *stdev_server* and *stdev_client*. *stdev_server* calculates a set of values including *sum2, mean* and *size*, while *stdev_client* aggregates these values to compute the overall standard deviation value of the decomposed dataset. To use these functions in a standard deviation assertion, a user issues the following commands in Guard:

```
register <path_to_C_file>/stdev.c
assert stdev ($a::var@par.c:100) > 0.2
```

The file *stdev.c* is compiled, and the *stdev* function is registered. The execution of the assertion computes the standard deviation value using each element in the array *var* at line 100 of source file *par.c*, and then compares it against the constant 0.2. If elements in array *var* deviate more than 0.2, the assertion fails and the program will be stopped for further examination.

4.2. User-defined Data Models

To help users create abstract data models, we introduce a new debugger built-in function, called *randset*, that defines random variates. The type of probability density function, such as Gaussian, Cauchy, Poisson etc, and the

number of samples, describe a random dataset. In addition, different distribution functions require different set of parameters including mean, standard deviation, and scale parameter values.

randset (<distribution_name>, <dataset_size>, …)

Currently, Guard supports various typical distribution models including Binomial, Gaussian, Cauchy, Poisson, and Maxwell-Boltzmann. The population of the random variates are created in parallel by the debug back-ends.

4.3. Model Comparison

With statistical assertions such as the histogram assertion, we do not compare two sets of exact values, but rather compare two abstract data models. Therefore, the differences can only be estimated via the χ^2 *goodness of fit test* [6]. For comparison of two histograms, the statistic χ^2 can be calculated using the general formula (3) [6].

$$\chi^2 = \sum_{i=1}^{N} \frac{(H_i - E_i)^2}{E_i} \text{ where } \begin{cases} H_i: \text{ the observed frequency for bin i} \\ E_i: \text{ the theoretical frequency for bin i} \end{cases} \quad (3)$$

The value of χ^2 then will be used to determine the *p-value* by comparing to a chi-squared distribution. The p-value is used to assert the hypothesis by comparing with user-defined *significance level α*. This parameter is also specified in the assertion. Consider the following Guard's specification:

```
create $model=randset (gaussian, 100000 ,0.05)
set reduce histogram (1000, 0.0, 1.0)
assert $a::my_array@code.c:455 ~ $model < 0.02
```

The example above describes the comparison of the histogram constructed using data obtained from the variable *my_array* at line 455 of source file *code.c*, and the histogram generated using the dataset pointed by the debugger variable *$model*. According to the *create* command, *$model* is a random variate consisting of 100,000 samples from the Gaussian distribution with standard deviation of 0.05. If the χ^2 test result is smaller than the significance level α=0.02, the hypothesis is accepted and the runtime data at that breakpoint follows the Gaussian distribution model.

4.4. Guard's Architecture

This section describes the general process of executing a statistical assertion through the current architecture. Guard employs a client/server model, where the debugger is divided into a single front-end client and multiple servers, to ensure that the processes being debugged can be distributed onto multiple processors and can be controlled independently (Figure 2). Communication between client and servers is performed by highly scalable network infrastructure such as MRNet [7]. The tree layout is explicitly determined when the debugger invokes a parallel program based on the number of processes involved. The client process is attached to the root node of the tree while debug servers are attached to the leaf nodes. Other parallel debuggers, such as TotalView [4], P2D2 [8], and DDT [9], use a similar client/server architecture, albeit with different server technology. For example, DDT uses GDB as its debug engine, whereas TotalView has its own proprietary code.

The details of how this architecture is useful for processing ad hoc debug-time assertions is discussed fully in our previous work [1, 2]. The following sections discuss the mechanism used to parallelise statistical assertions through two main aspects of this architecture: (1) the dataflow engine, which is the core engine for running ad hoc debug-time assertions [1], and (2) the functionalities of the statistic API which parallelises the statistical operations.

4.4.1. Dataflow Engine

An assertion in Guard is compiled into a low-level graph description upon creation [2], and is executed by a special interpreter. Figure 3 illustrates the execution flow of a statistical assertion. As described earlier, a statistical assertion involves the use of either a user-defined function or the built in reduction command. When the assertion is compiled, this information is encapsulated along with the variable's name (e.g via GETVAR_SET) and sent to debug servers. Debug servers reduce the raw data into primitive statistics in parallel according to the *server* function (1). The results are collected and passed up the network tree. These are integrated into the required statistic using the

client function by the front-end (2). The overall statistical measure is then compared against user's expectation (3) to flag the final outcome of the assertion.

4.4.2. Statistics API

A new component, *Statistic API*, is added into the current architecture, shown in Figure 2. This API contains a collection of functions, which validate, compile and communicate user-defined statistical functions between the front-end client and the back-end debug servers. The *register* command, discussed in 4.1.1 makes use of these API functions. Likewise, the debug servers invoke functions in this API to perform various statistical reduction activities. Furthermore, when the assertion requires the creation of user-defined data models, statistic API provides routines to evaluate the *randset* command and produce the random variates.

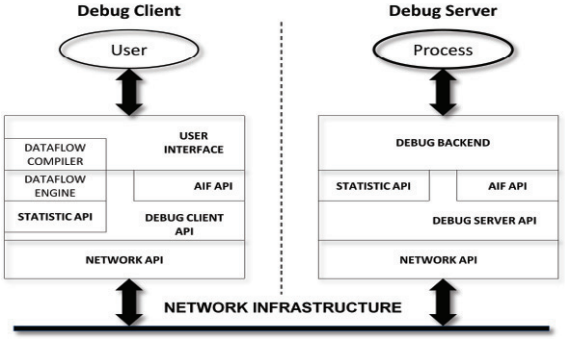

Figure 2 - Guard Layered Architecture With Statistic API

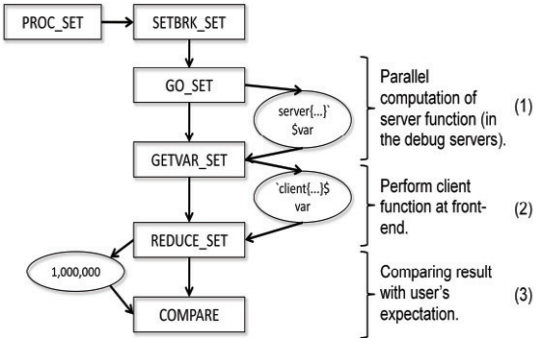

Figure 3 - Abstract Execution Of Statistical Assertion

5. Case Study: Molecular Dynamics

5.1. Background

In this case study, we used a parallel molecular dynamics code [10] written in C using MPI. We replicate a set of program bugs presented in Frenkel et al. [11] to illustrate the expressive power and potential of statistical assertions for finding errors. This code uses the Lennard-Jones (LJ) potential in modelling a fluid, which is popular for investigating various liquid phenomena such as melting, the liquid-vapour surface and nucleation [12], and is a fundamental simulation in molecular dynamics. The simulation consists of a 3D cube with each dimension of size L, which contains randomly positioned particles, with random initial velocities. At each time step, the system computes the new positions for all particles using the interaction force between themselves, and their current velocities.

5.2. Monitoring Particles' Speed

Since all particles have the same mass, their kinetic energy is only dependent on their speed. In any given fluid, the speed varies a great deal, from very slow particles to very fast ones. However, this scalar value spreads according to Maxwell-Boltzmann distribution [11]. Therefore, monitoring particles' speeds can help detecting anomalies in a simulation. In order to test this assumption, we assert that the histogram constructed using the *speed_array* variable is similar to a histogram generated with samples picked from Maxwell-Boltzmann distribution. This is done with the *histogram assertion* below:

```
create $model=randset (maxwell, 49152, 6, 2)
set reduce histogram (100,0.0,10.0)
assert $a::speed_array@pmd.c:28 ~ $model < 0.02
```

The first command builds a set of 49,152 samples from the Maxwell-Boltzmann distribution. This command

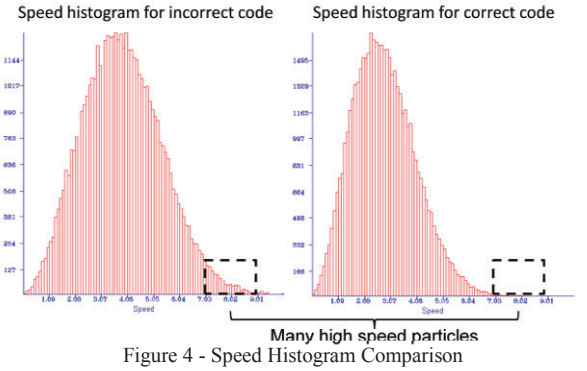

Figure 4 - Speed Histogram Comparison

requires two constants, which are the current temperature value *T*, and the particle's mass *m*. The distribution function is described in equation (4). The following commands request the data obtained from both the *speed_array* variable and the random number set to be reduced to histograms with 100 bins and accumulate data ranging from 0.0 to 10.0. This assertion fails after several simulation cycles. Figure 4 reveals a large number of excessively fast particles, comparing between the buggy behaviour and the expected behaviour. This anomaly is typically a result of using large time step (Δt) [13]. In this program, Δt is configured manually as a constant for the life of the simulation, and should be sufficiently small to make this behaviour unlikely. Rather, we suspect Δt has been misused where acceleration variable and velocity variable are computed. Closer investigation of the implementation of Velocity-Verlet algorithm revealed an error (shown below) where the first *Verlet's half-kick* is performed. Correcting the code fixes this bug.

$$f(v) = \sqrt{\frac{2}{\pi}\left(\frac{m}{kT}\right)^3} v^2 \exp\left(\frac{-mv^2}{2kT}\right) \text{ where : } k \text{ is Boltzmann constant} \quad (4)$$

*rv[i][a]=rv[i][a]+DeltaT*ra[i][a];*	Incorrect code
*rv[i][a]=rv[i][a]+DeltaTH*ra[i][a];*	Correct code

5.3. Energy Conservation

The law of *energy conservation* states that the total amount of energy in an isolated system remains constant. In a LJ system, *forces* are time independent, thus the *total energy*, which is the sum of *kinetic* and *potential* energies, should stay approximately constant [12], compared to the initial provided total energy (via initial temperature). Therefore, a drift of this quantity may signal programming errors. To detect this we trigger a *standard deviation assertion* after each simulation cycle to ensure this quantity does not alter significantly.

assert stdev ($a:: totEnergy@pmd.c:28) < 0.1

The assertion above inserts a breakpoint at line 28 where a simulation step is just completed. When it is executed, the debugger extracts the *totEnergy* variable, which holds all total energy values obtained so far, and performs the standard deviation operation. After a few simulation steps, the assertion is violated, indicating that total energy value has drifted. Since total energy is the sum of kinetic and potential energies, we inspect the code where those variables are computed. Importantly, because we keep a fixed temperature throughout the simulation, even though kinetic and potential energies do not stay absolutely constant; they are expected to oscillate around a constant value [12]. However, monitoring the potential energy variable for a few cycles, we notice that it does not fluctuate, but rather keep increasing. The potential energy arises from the interactions of particles with each other. Therefore, we inspect the segment of code where interaction force is computed. We realise the *force calculation function* is in error. According to the force formula (5), the programmer has missed the *ri2* term (i.e r^2) in the code.

$$f_x(r) = \frac{48x}{r^2}\left(\frac{1}{r^{12}} - 0.5\frac{1}{r^6}\right) \quad (5)$$

*fcVal=48*ri6*(ri6-0.5)+Duc/r1;*	Incorrect code
*fcVal=48*ri2*ri6*(ri6-0.5)+Duc/r1;*	Correct code

5.4. Summary

To date, we have shown how scientific knowledge can be converted into ad hoc debugging assertions in order to monitor the progress of the target program. Even though the assertions did not directly locate the defects in the source code, they successfully highlighted the anomalies in the progress of the simulation. This information was sufficient for the user to scope the region of the defective code and identify the errors. In addition, we demonstrated the usage of various statistical assertions including standard deviation, and histogram assertions. The following section presents a performance analysis and illustrates the practicality of the scheme.

6. Performance Analysis and Evaluation

In this section, we evaluate the performance of the *standard deviation assertion* on a 12,000 core Cray XE6 Gemini 1.2 system. To demonstrate that the approach is applicable to large-scale scientific codes, we use information about data structure sizes in a real molecular dynamics simulation. Branicio et al. investigated the atomistic mechanisms of fracture accompanying structural phase transformation (SPT) in AlN ceramic under hypervelocity impact using a MD simulation with a $209x10^6$ atoms [18]. Each atom requires 3 double precision floats to represent each of various keys attributes such as the velocity and acceleration, so the simulation requires 627 million double-precision floats per attribute. Using these parameters, we measures the scalability of the scheme as the number of CPU cores increases. We keep the size of the overall data structure constant as the number of processors changes; thus data distributed to each worker reduces as the number of processes increases.

Evaluating a *statistical assertion* consists of a few phases. First, after receiving the command from the front-end, the remote debug servers obtain data from GDB, and perform the requested statistical operation. This step proceeds in parallel since each debug server works independently. Through the network infrastructure (i.e. MRNet), data is reduced and returned to the debug client. We denote the time spent on this step as the *parallel reduction and communication* time. Then, the front-end executes *post data processing* tasks, performing the final reduction sequentially. Finally, the front-end performs *comparison* tasks to flag the outcome of the assertion.

Figure 5 shows these times along with the *overall assertion time* on a log scale. It can be seen that the overall assertion time is dominated by the *parallel reduction and communication* time, they almost overlap each other. This measure decreases as the number of processors increases because the parallel reduction time outweighs the communication time, due to the efficient aggregation of data throughout the MRNet communication network. The *post data processing time* grows as the number of processors increase, because it is performed sequentially in the front-end. Therefore, at around 8,192 cores, *parallel reduction and communication time* diverges from the *overall assertion time*. However, it is very small and hardly affects the overall assertion time. As a result, the tool achieves a good speedup overall as depicted in Figure 6, and more importantly, reduces the assertion execution time to the order of seconds, making is feasible as an interactive debugging aid.

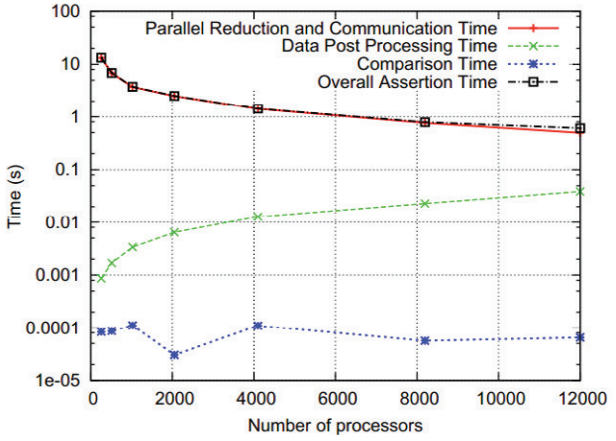

Figure 6 - Strong Scaling Standard Deviation Assertion

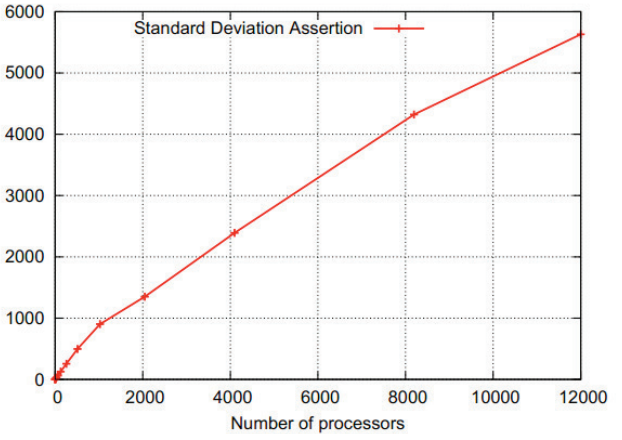

Figure 5 - Assertions Speedup Against #Processors

7. Related Work

The work discussed in this paper is a good example of Zeller's more general [15] *scientific debugging* method, in which a user invents a *hypothesis* about program behaviour, and then tests it against an *observation*. Related work on the use of statistical hypothesis includes Zhou et al. [16], Daikon [17], and DIDUCE [18]. They demonstrate that *statistics-rule-based* approaches are very promising in detecting bugs that do not violate any programming rules. However, they only work with sequential programs and are not evaluated in parallel. DMTracker [19] also employs the statistics-rule-based technique and provides a solution for parallel applications. The tool can automatically detect the cause of phenomena such as data corruption or deadlocks by observing data movements between parallel processing threads. A more notable example of applying statistics for debugging is the S*tatistical Debugging*

technique developed by Liblit et al. [20]. The author argues that stochastic failures can be reported multiple times and the information, extracted from reporting data via various statistical and modelling techniques, can be used to deduce the likely location of the bugs. However, the goal of DMTracker and statistical debugging technique is only to isolate a certain class of bugs namely *program runtime failure*. They obviously cannot be used to identify bugs that do not abort the operation of the program but silently corrupt the final results.

For interactive debugging paradigm, DDT [9] supports the use of a simple bar code like snapshot to show the spread of values for a given variable, while TotalView debugger [4] provides users with a limited number of statistical functions including max/min, mean, median, standard deviation, quartiles: first and third, and upper/lower adjacent. These are statistical functions of the type we envisage. In addition, TotalView's *TVScript* allows users to perform arbitrary actions at breakpoints, similar to the GDB's *conditional breakpoints* [21]. These are promising tools for monitoring statistical features of the application. However, with these tools, it is difficult to reason about the collective state of a parallel program. Our implementation in this work provides a parallel solution implicitly.

The *split-phase* scheme implemented in this work resembles the *MapReduce* programming paradigm [22]. In MapReduce paradigm, the *Map* step involves the partition of sub-problems onto multiple worker nodes, and these worker nodes perform the assigned tasks in parallel before passing the answer back to its master node. These activities are carried out by the *Parallel* phase in our scheme. Second, both the *Reduce* step in MapReduce and the *Aggregation* phase in the split-phase scheme take the answers to all the sub-problems and combine them in some way to get the output. It would be interesting to evaluate whether existing MapReduce run times could be used to support this work, although, the infrastructure for parallel evaluation and reduction already exist in Guard.

8. Conclusion and Future Work

Regardless of the programming paradigm, debugging tools typically report raw data values. This approach has become impractical when there are very large data structures, particularly distributed over large parallel computers. Typical scientific codes have enormous multi-dimensional data structures and it is not viable to expect a user to trace problems without using further data analysis or data reduction methods. In addition, beside the actual source code, other knowledge about the application is not typically used to reason about the progress of the target program. We believe that statistical techniques help address some of these issues.

The earlier work on ad hoc debug-time assertions is the foundation for the work in this paper. Here, however, we have demonstrated that statistical information instead of raw data, can be very useful in the debugging process. Moreover, this can be accelerated using a parallel computer, and we demonstrated this on up to 12,000 cores. Specifically, we showed on some reasonably sized applications that complex assertions can be executed in the order of seconds, making the technique feasible for real time debugging.

Spotting outliners is a crucial part of debugging, regardless of the exact methodology used. However, when data structures get large and complex, finding values that stand out from the rest becomes increasingly difficult. Modern data mining techniques allow user to quickly cluster data values and identify outliners. As a result, we plan to incorporate those techniques into our debugging tool, so outliners can be found in massive datasets, which leads to the isolation of potential bugs.

Acknowledgements

This project is supported by the Australian Research Council under the Linkage grant scheme, and is supported by Cray Inc. We acknowledge our colleagues in the MeSsAGE Lab at Monash University. This material is based upon work supported by the Defence Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defence Advanced Research Projects Agency. The project is also supported by the Office of Science of the U.S. Department of Energy under Agreement No. DE-FG02-09ER25929.

References

[1] M. N. Dinh, D. Abramson, D. Kurniawan, C. Jin, B. Moench, and L. DeRose, "Assertion based parallel debugging", in *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Newport Beach, California, 2011.

[2] G. R. Watson, "The Design and Implementation of a Parallel Relative Debugger", in *Faculty of Information Technology*. vol. PhD Thesis Melbourne: Monash University, 2000, p. 197.

[3] T. F. Chan, G. H. Golub, and R. J. LeVeque, "Algorithms for computing the sample variance: analysis and recommendations", *The American Statistician,* vol. 37, pp. 242-247, 1983.

[4] ETNUS, "Discovering TotalView", 2003. Retrieved 29/10/2010, from http://www.jaist.ac.jp/iscenter-new/mpc/old-machines/altix3700/opt/toolworks/totalview.6.3.0-1/doc/html/user_guide/intro.html

[5] Python Software Foundation, "Python Programming Language", 2011. Retrieved 18/07/2011, from http://www.python.org/

[6] Yale University, "Chi-Square Goodness of Fit Test", 1998. Retrieved from http://www.stat.yale.edu/Courses/1997-98/101/chigf.htm

[7] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-based Multicast/Reduction Network for Scalable Tools", in *Proceedings of SC*, Phoenix, AZ, 2003.

[8] D. Cheng and R. Hood, "A Portable Debugger for Parallel and Distributed Programs", in *Conference on High Performance Networking and Computing*, Washington, D.C., United States 1994, pp. 723-732.

[9] Allinea, "Allinea DDT – A revolution in debugging", 2009.

[10] CACS, "HIGH PERFORMANCE COMPUTING AND SIMULATIONS", 2011. Retrieved 16/01/2011, from http://cacs.usc.edu/education/cs653.html

[11] D. Frenkel and B. Smit, *Understanding Molecular Simulations: From Algorithms to Applications*, 2 ed. Elsevier Science & Technology, 2002.

[12] J. J. Nicolas, K. E. Gubbins, W. B. Streett, and D. J. Tildesley, "Equation of state for the Lennard-Jones fluid", *Molecular Physics,* vol. 37, issue 5, pp. 1429-1454, 1979.

[13] W. Wieser, "Simple molecular dynamics simulation of a Lennard-Jones fluid", 2008. Retrieved from http://www.triplespark.net/sim/ljfluid/

[14] P. S. Branicio, R. K. Kalia, A. Nakano, and P. Vashishta, "Shock-Induced Structural Phase Transition, Plasticity, and Brittle Cracks in Aluminum Nitride Ceramic", *PHYSICAL REVIEW LETTERS,* vol. 96, issue 6, 2005.

[15] A. Zeller, *Why programs fail: A guide to systematic debugging*. Elsevier, 2006.

[16] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatically Detecting Memoryrelated Bugs via Program Counterbased Invariants", *37th International Symposium on Microarchitecture (MICRO),* pp. 269-280, 2004.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 27, issue 2, 2001.

[18] S. Hangal and M. S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection", in *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, 2002.

[19] Q. Gao, F. Qin, and D. K. Panda, "DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements", in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Reno-Tahoe, USA, 2007.

[20] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, "Statistical Debugging of Sampled Programs", in *Neural Information Processing Systems (NIPS 2003)*, Vancouver, British Columbia, 2003.

[21] Free Software Foundation Inc, "GDB: The GNU Project Debugger", 2008. Retrieved 15/01/2009, from http://www.gnu.org/software/gdb/

[22] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", in *Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004.